Program Instrumentation and Trace Analysis

Klaus Havelund
Allen Goldberg
Kestrel Technology
NASA Ames Research Center
California, USA

Robert Filman
Grigore Rosu
RIACS
NASA Ames Research Center
California, USA

Several attempts have been made recently to apply techniques such as model checking and theorem proving to the analysis of programs. This shall be seen as a current trend to analyze real software systems instead of just their designs. This includes our own effort to develop a model checker for Java, the Java PathFinder 1, one of the very first of its kind in 1998. However, model checking cannot handle very large programs without some kind of abstraction of the program. This paper describes a complementary scalable technique to handle such large programs. Our interest is turned on the observation part of the equation:

How much information can be extracted about a program
from observing a single execution trace?

It is our intention to develop a technology that can be applied automatically and to large full-size applications, with minimal modification to the code. We present a tool, Java PathExplorer (JPaX), for exploring execution traces of Java programs. The tool prioritizes scalability for completeness, and is directed towards detecting errors in programs, not to prove correctness. One core element in JPaX is an *instrumentation* package that allows to instrument Java byte code files to log various events when executed. The instrumentation is driven by a user provided script that specifies what information to log. Examples of instructions that such a script can contain are: "report name and arguments of all called methods defined in class C, together with a timestamp"; "report all updates to all variables"; and "report all acquisitions and releases of locks". In more complex instructions one can specify that certain expressions should be evaluated and even that certain code should be executed under various conditions. The instrumentation package can hence be seen as implementing Aspect Oriented Programming for Java in the sense that one can add functionality to a Java program without explicitly changing the code of the original program, but one rather writes an aspect and compiles it into the original program using the instrumentation.

Another core element of JPaX is an *observation* package that supports the analysis of the generated event stream. Two kinds of analysis are currently supported. In *temporal* analysis the execution trace is evaluated against formulae written in temporal logic. We have implemented a temporal logic evaluator on finite traces using the Maude rewriting system from SRI International, USA. Temporal logic is defined in Maude by giving its syntax as a signature and its semantics as rewrite

equations. The resulting semantics is extremely efficient and can handle event streams of hundreds of millions events in few minutes. Furthermore, the implementation is very succinct. The second form of even stream analysis supported is *error pattern* analysis where an execution trace is analyzed using various error detection algorithms that can identify error-prone programming practices that may potentially lead to errors in some different executions. Two such algorithms focusing on concurrency errors have been implemented in JPaX, one for deadlocks and the other for data races. It is important to note, that a deadlock or data race potential does not need to occur in order for its potential to be detected with these algorithms. This is what makes them very scalable in practice. The data race algorithm implemented is the Eraser algorithm from Compaq, however adopted to Java. The tool is currently being applied to a code base for controlling a space craft by the developers of that software in order to evaluate its applicability.